Scalable and Manageable Customization of Workflows in Multi-Tenant SaaS Offerings

Majid Makki Dimitri Van Landuyt Stefan Walraven
Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
{majid.makki,
dimitri.vanlanduyt,
stefan.walraven,
wouter.joosen}@cs.kuleuven.be

ABSTRACT

In cloud computing, multi-tenancy is concomitant with scalability in the sense that sharing a single deployment instance between many customer organizations (tenants) maximizes the utilization of the available resources. However, this also introduces the need to customize the application to the (slightly) different requirements of different tenants. In the context of workflow-based SaaS offerings, this is not straightforward to accomplish without compromising scalability or manageability of the offering.

In this paper, we present a middleware for multi-tenant customization of workflows that enables software providers (i) to decrease coupling of multi-tenant customization concerns and workflow design for better manageability, (ii) to activate tenant preferences at runtime, but (iii) without incurring a scalability penalty. We validate a prototype implementation of our middleware in the context of a realistic SaaS application, evaluate its scalability, and extensively compare the results with related work confirming that the proposed middleware indeed supports customization of workflows in a significantly more scalable fashion.

CCS Concepts

•Information systems → Middleware business process managers; •Networks → Cloud computing; •Software and its engineering → Software as a service orchestration system;

SAC 2016, April 04-08, 2016, Pisa, Italy

DOI: http://dx.doi.org/10.1145/2851613.2851627

Keywords

Workflow-based SaaS, BPaaS, Multi-tenancy, Business Process Customization, BPMN 2.0, RedHat JBoss jBPM

1. INTRODUCTION

Software-as-a-Service (SaaS) is a software delivery model which is becoming very popular. This model is most efficient when offered in a multi-tenant fashion. Multi-tenancy improves economies of scale by helping user organizations to reduce their costs by sharing the whole computational stack including network infrastructure, CPU, memory, storage devices, operating system, application server and finally application instance among multiple customer organizations, a.k.a. tenants. However, requirements of tenants are not identical. Hence, when sharing an application instance, they have to be able to customize the application according to their business-specific requirements. Customizing a multi-tenant application, however, is different from customizing a classic (i.e. single-tenant) application in that the latter leads to deployment of a new application instance while the former keeps the customization of all tenants isolated in a single application instance without redeploying it. In addition to that, many SaaS applications are workflow-based or represent a business process (i.e. BPaaS[1]) which adds to the complexity.

Multi-tenant customization is required at the level of workflow definition which is more complex than customizing software by substituting service calls (i.e. an issue extensively investigated in both web service and cloud computing communities [7, 9, 15, 19, 21, 25, 26]) because there are more elements involved in a workflow definition. The existing solutions for business process customization in a multi-tenant context either sacrifice *manageability* (e.g. [8, 22]) or *scalability* (e.g. [14]). The main challenges of customizing a workflow-based multi-tenant offering are keeping software manageable for both software providers and tenant administrators with minimal impact on scalability.

Making a multi-tenant offering customizable can pose a man-

[1]For the sake of simplicity, we no longer refer to BPaaS. But every statement about workflow-based SaaS holds for BPaaS as well.

ageability problem for both software providers and tenant administrators. If software variability is coupled with the workflow definition (e.g. as it is the case in [8]), high variance in requirements of different tenants leads to an overtly complex process definition which is difficult to understand and maintain. This is initially a problem for software providers in charge of managing software over time. Additionally, it can lead to a manageability problem on the side of tenants. Multi-tenant offerings have to be configurable in a self-service manner without any human intervention by the software provider because software providers usually increase their revenue mainly by targeting a lot of small and medium businesses instead of a few large enterprises who can afford on-premise deployments. Therefore, if the tenant administrators are faced with a workflow definition including a lot of elements which are irrelevant for them but relevant for other tenants (e.g. as it is the case in [22]), it is difficult for them to go through the self-service configuration wizard without help from the staff members of the software provider. In that sense, coupling of multi-tenancy concerns with the workflow definition is also a manageability problem for tenant administrators.

The second challenge is minimal impact on scalability. If the use of computational resources by the customization solution increases enormously when the number of tenants increases, the solution is in fact sacrificing scalability. For instance, the solution proposed in [14] loads a new variant of the process definition in memory for each tenant. Such a memory overhead sacrifices scalability and consequently contradicts the main purpose of multi-tenancy which is reducing costs by sharing the computational stack among many tenants. The way a workflow-based SaaS offering is made customizable should have minimal impact on the way the consumption of computational resources increases with an increasing number of tenants.

In this paper, we present a middleware for business process customization in a multi-tenant context. As opposed to existing solutions, our middleware neither sacrifices ***manageability*** nor ***scalability***. The contributions of this paper are two-fold: (i) increasing manageability by considerably reducing the coupling between multi-tenancy and customization concerns on the one hand and business process definition on the other; (ii) avoiding load of a workflow definition variant in memory for each tenant by late binding of tenant configurations and consequently preserving scalability to a significant extent.

As a proof-of-concept, we have implemented the middleware based on BPMN 2.0 [1] and validated its functionality in the context of a realistic document processing SaaS application to see if it truly enables workflow customization for each tenant in isolation from others. Even though our implementation is based on BPMN 2.0, we explain why and how the same approach is applicable to other business process definition languages. We have also evaluated the scalability of our solution in comparison with the most important existing solution [14]. We show that our middleware is significantly more scalable. We also present evaluation results confirming that the performance overhead of the middleware is negligible. In an analytical discussion, we demonstrate in what respects our middleware improves manageability of the SaaS offering.

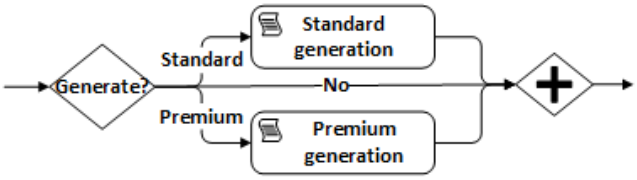

Figure 1: Document processing workflow fragment with multi-tenancy concerns embedded in the process definition.

In the next section, we elaborate on the problem more substantially with an example. We present the design of the middleware in Section 3. Functional validation of multi-tenant customization and evaluation of scalability and performance are presented in Section 4. In Section 5, we discuss several aspects of the work including the manageability issue. We contrast our solution with related works in Section 6. Finally, the paper concludes in Section 7.

2. MOTIVATION

The motivating example for this paper is taken from a real-world industrial SaaS application, a document processing system that is inherently workflow-driven. This workflow includes a document generation phase. Some tenants may choose to provide generated documents and consequently skip over it. A few tenants can afford the premium generation, which offers more flexible and beautiful templates, while others may choose a standard generation phase which is less expensive. This phase of the workflow is depicted in Figure 1 which is designed by what the state-of-the-practice offers. In this approach, all the options are in one workflow definition and decision points are added to handle tenant preferences.

The problem of this design is that with increase in the number of workflow steps and divergence in requirements of tenants, the process definition explodes. Such an explosion is undesirable for at least two reasons. Firstly, managing such a complex and overtly-branched process definition is too difficult for the SaaS providers. Secondly, tenants, while configuring the application, face a process definition which includes a lot of elements that are irrelevant for them. For instance, why should a tenant see a parallel gateway in the workflow definition which is there only to converge the flow which is branched only because there are other tenants with different requirements? This problem remains unaddressed in some existing works (e.g. [8, 22]).

One solution, proposed by Mietzner et al. [14], is to have a template process definition with certain points of extension which can be filled in according to the choices made by tenant administrators. Choices are regulated according to variability descriptors which are orthogonal to the process definition. Hence, variability is loosely coupled with workflow design. However, filling in the template follows by loading a new variant of the process definition in the workflow engine. Loading a new process definition in the workflow engine for each new tenant entails enormous in-

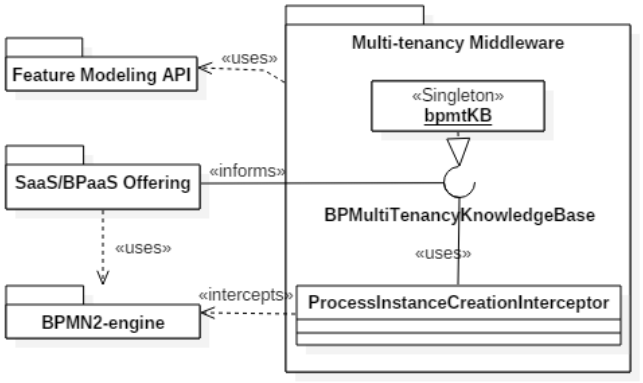

Figure 2: Building blocks

crease in memory usage. In short, memory usage in this approach increases enormously with increase in the number of tenants; hence, significant reduction of scalability.

3. MIDDLEWARE DESIGN

In this section, we present the middleware which addresses the above challenges. First we explain the principal design decisions and subsequently we present a detailed view of the middleware.

3.1 Principal Decisions

The design of the middleware is founded upon three cornerstones: (i) systematic management of software variability (cf. [12]), (ii) 'underspecification' of process definition (cf. [23]) and (iii) late binding of tenant configurations.

Variability Management. In order to systematically manage variants of the workflow definition for different tenants, we employ feature modeling (cf. Kang et al. [12]). A feature model is a hierarchical view of a variable software application consisting of coherent functional units called features. Features can be optional/mandatory and abstract/concrete. Sibling features can be composed together or be each other's alternatives. It is also possible to impose constraints on possible compositions of features. Based on a feature model, features can be composed to make a software configuration for each tenant. Abstract features cannot be part of a configuration but their concrete children can represent them in a configuration. We consider any **abstract** feature with **alternative** sub-features and/or any **optional** feature a variation point. A variation point is where preferences of tenants play a role.

Process Underspecification. The variation points can be anywhere in the process definition in form of points for diverting the control flow from the master process to sub-processes. These points are variable placeholders and the variable in one of these points is supposed to point to a sub-process. Sub-processes can consist of any element and in any order and composition which are allowed for normal processes. Since a sub-process is a coherent functional unit, it is the best match for a feature in the feature model. Since the points for diverting the control flow from the master process to sub-processes are variable placeholders, the master process is 'underspecified' (cf. Van der Aalst [23]) at design time. Hence, it needs to be fully specified at runtime.

```
Data: BPMultiTenancyKnowledgeBase kb
Input : Process p, VariableValue[] vals
 2 if kb.rootFeature.name == p.name then
 3     firstInstantiation = true;
 4     for each variable v in p do
 5         if v == '_tenantConfig' then
 6             firstInstantiation = false;
 7         end
 8     end
 9     if firstInstantiation then
10         p.addVariable('_tenantConfig');
11         for each node n in p do
12             featureName = n.calledElement;
13             if n is CallActivity && kb.featureNames
               contains featureName then
14                 n.calledElement =
                   '#{_tenantConfig['';
15                 n.calledElement += featureName;
16                 n.calledElement += '']}';
17             end
18         end
19     end
20     tenantId = vals['_tenantId'];
21     vals['_tenantConfig'] =
       kb.configs[tenantId];
22 end
23 proceed to jBPM;
```

Algorithm 1: Around advice

Late Binding of Configurations. Activating tenant configurations amounts to fully specifying the 'underspecified' master process definition. This is done by assigning an appropriate value to the variable in charge of determining which sub-process should be invoked at every variation point for a specific tenant. The middleware intercepts the workflow-engine at the moment of creating a process instance based on the master process definition in order to activate a tenant-specific configuration. This late binding of tenant's configuration helps us avoid multiplying the process definition in memory for each tenant and consequently avoiding scalability repercussions.

3.2 Detailed View

Since BPMN 2.0 has already the notion of sub-process and a mechanism for diverting the control flow to sub-processes, we choose this process modeling language for our proof-of-concept. In BPMN 2.0, *CallActivity* nodes enable diversion of the execution from a parent process to sub-processes by means of values assigned to their *calledElement* attributes. Even though this mechanism is intended to enable process reusability according to BPMN 2.0 [1], we use it to implement the aforementioned 'underspecification' and late binding techniques because it is also possible to assign a variable to *calledElement* and later assign a value to that variable. The same approach can be adopted for some other workflow definition languages. For instance, in Oracle or IBM implementations of BPEL, it is possible to invoke another

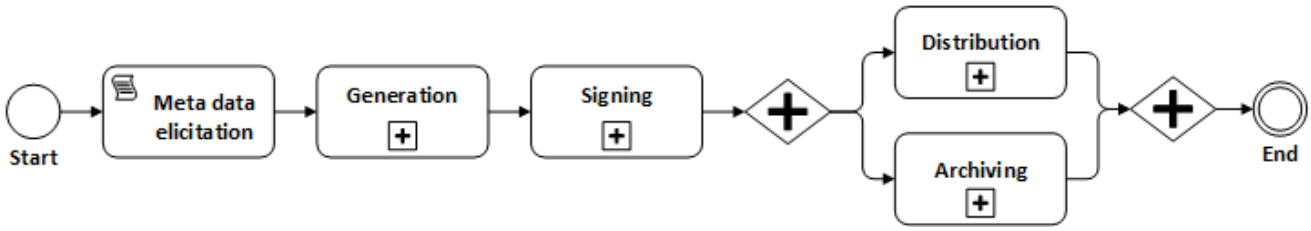

Figure 3: Master process definition for document processing SaaS offering

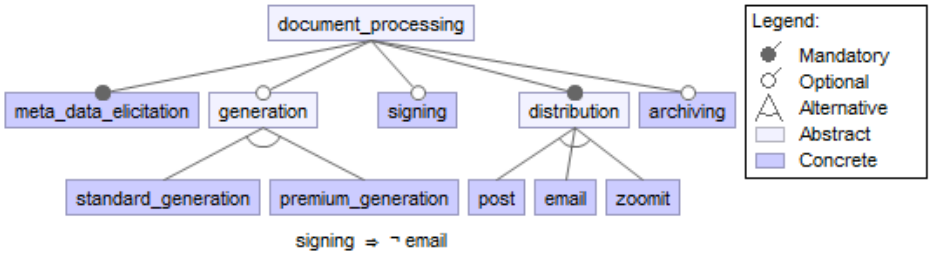

Figure 4: Feature model (created by FeatureIDE [20]) for document processing SaaS offering

process, i.e. a sub-process, in a parameterized manner.

Figure 2 shows the building blocks of the middleware in relation to BPMN2-engine (we use jBPM [2]), a feature modeling API (we use the core package of FeatureIDE [20]) and the multi-tenant offering. The middleware consists of two main components. The first component is a singleton object where the feature model is stored along with the configuration of each tenant. A tenant configuration is a set comprising of the name of concrete features selected by the tenant administrator. This singleton object is an instance of the Java interface *BPMultiTenancyKnowledgeBase* which exposes methods for initializing, updating, getting the feature model on the one hand and adding, removing, updating, getting tenant configurations on the other. The multi-tenant offerings use this programming interface to inform the middleware of the feature model and the tenant configurations. The second component, *ProcessInstanceCreationInterceptor*, is an AOP aspect in charge of intercepting jBPM at the moment of creating a process instance. We employ AOP techniques to intervene for two main reasons. Firstly because AOP is the best option for implementing cross-cutting concerns and multi-tenancy is a cross-cutting concern. Secondly because this decision helps us avoid any change in the code of jBPM.

Algorithm 1 shows how the around advice of *ProcessInstanceCreationInterceptor* works to modify the workflow based on tenant's configuration. Lines 3 to 8 check if it is the first time an instance of the master process is created. Lines 10 to 18 are responsible for adding a variable to the process definition which holds the tenant's configuration and also for adjusting the values assigned to *calledElement* attributes of *CallActivity* nodes corresponding to the variation points. It is sufficient to execute these lines once in the entire application lifetime because they operate at the level of process definition not the process instance.

On the contrary, the lines 20 to 21 are executed each time an instance of the master process is created by jBPM and operate at the level of process instance instead of the process definition. These lines make sure that the tenant's configuration is taken into account for diverting the flow from the master process to the sub-processes for a specific process instance by retrieving the tenant's configuration from the singleton object of *BPMultiTenancyKnowledgeBase*. The values retrieved from the singleton object of *BPMultiTenancyKnowledgeBase* in line 21 are in form of a mapping from feature name of a variation point to the id of a sub-process corresponding to a concrete feature selected by the tenant. These values are calculated for each tenant only at (re-)configuration time. Therefore, the algorithm for calculating these values does not impose any performance overhead to the execution of process instances. That algorithm traverses the tree of the feature model, finds variation points (i.e. abstract features with alternative sub-features or optional features) and pushes them into a map object as keys referring to the name of a concrete feature selected by the tenant.[2]

4. EVALUATION

In this section, we (i) validate the functionality of our middleware to see if it truly enables customization for each tenant in isolation, (ii) evaluate how scalable our solution is and compare it with the most important solution proposed so far, (iii) and finally, we present the performance overhead imposed by the middleware because our solution intercepts the workflow-engine for each process instantiation to avoid scalability repercussions.

[2]The middleware prototype along with test cases and evaluations are available on http://people.cs.kuleuven.be/~majid.makki/acm-sac-2016/main.html.

The evaluations are performed in the context of a document processing application inspired by the requirements of our industrial partner active in this market. The master process definition is depicted in Figure 3. The rectangles with a + at the bottom are *CallActivity* nodes corresponding to variation points visible in Figure 4. To repeat, variation points are optional features or abstract features with alternative children. Hence, *generation*, *signing*, *distribution* and *archiving* are variation points. We have designed and implemented the process definition of sub-processes corresponding to the concrete candidate features: *standard_generation*, *premium_generation*, *signing*, *post*, *email*, *zoomit* and *archiving*. But due to space limitation, we do not present them here. In addition to these sub-processes, there is a special sub-process called *skip* which fills in any variation point corresponding to an optional feature if the tenant decides to exclude it from their configuration. This sub-process is a start node immediately connected to a terminate node and helps skip over a *CallActivity* node corresponding to an optional feature such as *signing*.

The technical setup for evaluations is a Windows 8.1 (64-bit) machine with Intel Core i7 (3.6 GHz), 16 GB of memory, and JRE 1.8 of Oracle HotSpot 64-bit running a Java standalone application using jBPM 6.2 started by 8 GB for maximum JVM heap size. The heap size could be significantly lower but in order to measure the memory effect of the solution proposed by Mietzner et al. [14] for comparison, we used the aforementioned heap size.

4.1 Functional Validation

In this section, we validate the functionality of the middleware w.r.t. the purpose of customization in isolation by implementing two realistic scenarios.

The first scenario is a situation where a new tenant joins in and starts a new process instance while two already-existing tenants have undergoing process instances and one of them starts another process instance. Finally, without restarting/redeploying the application, all the process instances were executed according to the flows dictated by the configuration of each tenant which is the expected behavior.

The second scenario is a situation where *TenantA* and *TenantB* each have an ongoing process instance when *TenantA* changes its configuration and starts a new process instance. The process instance of *TenantB* proceeds, according to their specific configuration, without restarting/redeploying the application for reconfiguring *TenantA*. The same holds for the first process instance of *TenantA* which proceeds according to their initial configuration, i.e. expected flow before reconfiguration. And the second process instance of *TenantA* executes according to the flow determined by their new configuration. This second scenario shows that reconfiguration on the side of one tenant is possible without restarting/redeploying the application and without affecting the functional behavior of the undergoing process instances.

4.2 Scalability

Scalability of a multi-tenant offering is directly linked to the question of how resource consumption increases when the number of tenants increases. We can see in Algorithm 1 that

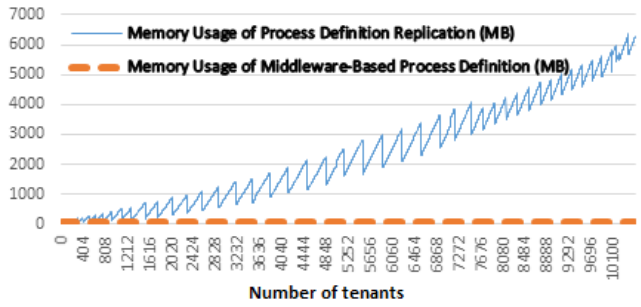

Figure 5: Scalability Comparison with the memory-intensive approach.

the worst-case complexity of the interceptor is $O(n+v)$ and $O(v)$ for the first instantiation of the master process and the subsequent instantiations respectively where n is the number of nodes in the master process and v is the number of variables in it. This proves that CPU-utilization is independent of the number of tenants. Hence, the middleware is highly scalable insofar as CPU-utilization is concerned. The same holds for memory usage of the middleware. When the number of tenants increases, memory usage will only increase slightly because of the additional tenant configurations loaded into memory, which is an inevitable characteristic of any multi-tenant software. In practice, this can be simply improved by lazy loading of tenant configurations from a database.

As mentioned before, an existing approach is to load a variant of the process definition in memory for each individual tenant (e.g. [14]). We call that the memory-intensive approach. We have compared the memory used by jBPM under two settings: (i) loading multiple times a tailored down process definition for a tenant choosing {*meta_data_elicitation, standard_generation, signing, post, archiving*} and (ii) loading once the master process (Figure 3) along with all the sub-processes including *skip*. The former is representative of the memory-intensive approach and the latter is representative of our solution. Memory usage in our case is 30.5 megabytes, and as shown in Figure 5, does not change with increasing number of tenants while the memory used in case (i) increases 400%, 1330%, 3168%, 7143% and 9611% when the number of tenants reaches 1000, 2500, 5000, 7500 and 10000 respectively.[3] It should be noted that these numbers do not include the memory used for serving requests. They are simply limited to the memory occupied by process definitions. Furthermore, this test case is limited to a workflow definition consisting only of script tasks. In more complex applications, timer and web-service configurations can increase memory usage a lot more for each variant of the process definition.

4.3 Performance Overhead

Even though the above comparison demonstrates the significant memory usage reduction brought about by our solution,

[3]The bounces in Figure 5 are the effect of the Java garbage collector (GC) which releases memory of unused objects. But even Java GC cannot stop the increasing memory usage.

	Multi-tenancy Middleware	jBPM
First Inst.	1.21 ms	91.87 ms
Subsequent Inst.	0.06 ms	19.94 ms

Table 1: Performance Overhead of Multi-tenancy Middleware for jBPM.

it is not evident at what cost this improvement is achieved. The main cost is the performance overhead of the interceptor thanks to which we avoid multiplying variants of the process definition in memory.

In order to measure this performance overhead, we firstly compare the time taken by our interceptor with the time taken by jBPM. The master process definition has 7 nodes and 3 variables. We used the following tenant configuration: {*meta_data_elicitation, standard_generation, email, archiving*}. Table 1 shows the computation time taken by our middleware versus that of jBPM for both creating the first process instance and the subsequent instantiations. When the master process is instantiated for the first time, the average computation time overhead of our interceptor is 1.32% of the time taken by jBPM itself over 10 attempts. This occurs only once in the entire application life-time. The performance overhead of the interceptor for subsequent instantiations of the master process which occur repeatedly is more important. The computation time overhead of the interceptor is, on average and in 10 attempts, 0.29% of the time taken by jBPM which is a negligible performance overhead.

Secondly, we compare the execution time of the document processing workflow using our middleware versus a single workflow design covering the requirements of all tenants with multi-tenancy concerns embedded in it in form of decision points (similar to Figure 1). Similarly, in the middleware-based implementation, the master process definition has 7 nodes and 3 variables. As depicted in Figure 6, for the first time instantiation of the process, execution by our middleware was found to be more than three times slower than the embedded setting on average after 10 attempts. However, this is absolutely negligible as it occurs only once in the entire application life-time. On the contrary, for the subsequent instantiations of the process, our middleware, after 10 attempts, improved performance up to 15% on average. There are two main reasons for this performance improvement. First, our middleware applies preferences of tenants only at one place just before creating the process instance while preferences of tenants are checked at every variation point by means of decision points in the embedded setting. The second reason is that our middleware tailors down the process trajectory according to the preferences of the tenant while the complete trajectory should be traversed in the embedded setting.

5. DISCUSSION

Manageability Cost for the Provider. SaaS providers ad-opting our approach benefit from clear traceability between the variability management view of their system and the business process modeling view of it. The concrete features of the feature model correspond to sub-processes and

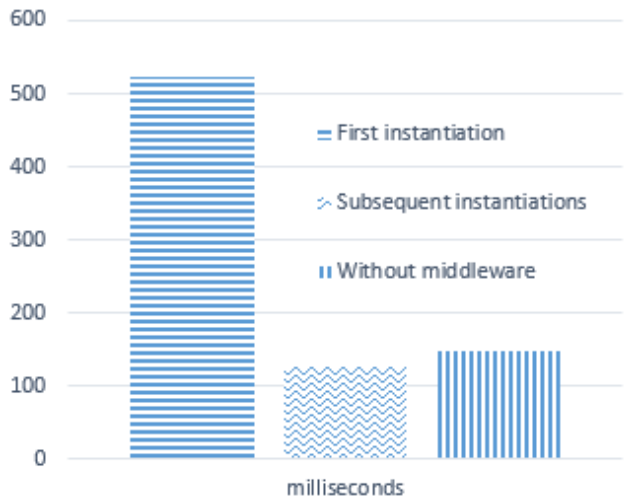

Figure 6: Workflow Execution Time with and without Multi-tenancy Middleware.

variation points correspond to *CallActivity* nodes of the master process. This way, they will have a more understandable code base which reduces the time and effort needed to maintain their offering over time.

Furthermore, our middleware helps them easily provide new variants of the system. Insofar as they want to add/remove a new concrete feature as a candidate for a variation point, they just need to add/remove a node to/from the feature model and a new sub-process definition to/from the workflow-engine.

However, there is also a management overhead. The providers have to maintain two more artifacts in their code base: a feature model and a group of sub-processes. Thus, the management overhead is a function of the number of features, f, and the total number of sub-process candidates for variation points, s. Fortunately, since the number of tenants, t, extremely exceeds both f and s (i.e. $t \ggg f$ and $t \ggg s$), the per-tenant engineering effort is still less than the effort needed in a classical software delivery model where a separate instance of the application is offered to each user organization and consequently even the shared parts of the code base should be maintained separately for each of them.

Finally, it is in favor of providers that our middleware can co-evolve with the underlying workflow-engine thanks to loose coupling made possible by the choice of AOP and the fact that there is only one pointcut which is advised.

Self-service Configuration for Tenants. Since our middleware helps software providers to exclude variability management elements from the process definition (e.g. control gateways in Figure 1 of Section 2), tenant administrators do not need to deal with elements in the process definition which are there only for handling multi-tenancy. Moreover, since tenant administrators start the self-service configuration wizard by looking at the master process from which all tenant-specific elements are abstracted, they are not perplexed by a lot of elements which are irrelevant for them and are only relevant for their neighbour tenants. They make

decisions about tenant-specific elements in the subsequent steps of the configuration wizard instead where the options are narrowed down according to the previous choices.

6. RELATED WORK

This work differs from the body of research on customization of business processes in that most of them target classic (i.e. single-tenant) software delivery settings. Works such as [3, 5, 10, 11, 17, 18, 24] are examples of this vastly investigated research domain. Firstly, their solutions are based on the assumption that software variability is solely managed by software providers. This does not hold in a multi-tenant context where tenants themselves are supposed to configure the application for themselves. Secondly, and more importantly, these solutions do not take into account the fact that customizing the business process for one tenant should be done in isolation such that the service is not stopped for other tenants.

A second research track that intersects with this work is multi-tenant software customization in general [7, 9, 15, 19, 21, 25, 26]. However, most of these works focus on substitutability of service calls. Since variation points in our work are sub-processes, any mixture of workflow elements such as service calls, timers, event catchers, user tasks, script tasks and control gateways can be bundled in a candidate for a variation point. Furthermore, execution of a sub-process can take days while a service call is supposed to terminate after a few seconds.

Mietzner et al. [15] highlight the need for workflow management in the context of customizable multi-tenant SaaS applications. However, they focus on offering predefined workflow definitions for customizing the behavior of the application while our focus is on customizing the workflow definition itself.

Among the works on business process customization in a multi-tenant context, the work by Mietzner et al. [14] is the most important so far. However, while they stress on the necessity to redeploy a new process definition for each tenant after substantiating a template process model in [14, 16], our solution avoids the enormous increase in used memory caused by this approach (cf. Section 4.2). Avoiding this makes multi-tenant offerings based on our middleware more scalable in the sense that significantly more tenants can be served by a single node in the network. Even though Mietzner et al. [14] have implemented their solution based on BPEL, we expect more or less the same results as presented in Section 4.2 because BPEL-engines also keep process definitions in memory. Scaling out on several nodes by means of a distributed memory API can come to rescue for the memory-intensive solution proposed in [14]. However, scaling out is always more costly than scaling up. Therefore, scalability is sacrificed to a considerable extent by that solution. Moreover, the existing workflow engines do not use distributed memory and a multi-tenancy middleware which requires the workflow engine to be built upon a distributed memory API cannot be easily adopted in practice.

Instead of a process definition covering the requirements of all possible tenants which is at the core of solutions proposed by Van der Aalst [22] and Gey et al. [8], a multi-tenant application based on our middleware deals with process definition and variability management as two separate design concerns. This separation of concerns makes software more manageable for software providers. It, furthermore, helps the tenant administrators in charge of configuring the application for their organization by allowing them to focus only on what is specifically relevant for them.

Even though the purpose of dynamic adaptation of workflows in [3] is not multi-tenant customization, our solution shares the technical approach with it. The latter solution also employs AOP techniques for intercepting the execution of workflow. However, it intercepts every step of the workflow execution while our middleware intercepts the workflow-engine only once for each process instance. Hence, we do not impose too big of a performance penalty for enabling dynamic behaviour.

Works such as [4, 6, 13], which add tags to BPEL for enabling AOP in it, are not specifically intended for enabling multi-tenant workflow customization. However, they can be adopted for that purpose. But these solutions are more difficult to adopt in practice because they extend the underlying process definition language syntactically. We have avoided making syntax changes to BPMN 2.0 by handling multi-tenancy concerns in an aspect outside of the process definition. This makes it easier for workflow-based SaaS providers to adopt our approach in practice.

7. CONCLUSION

This paper presented a solution for enabling customization of workflow-based SaaS applications for multiple tenants sharing the same application instance. Three design decisions leading to this solution are (i) separating variability management from process design by means of feature modeling, (ii) enabling specialization/generalization of workflows by a technique called 'underspecification', and finally (iii) activating tenant configurations at runtime by late binding of sub-processes to the 'underspecified' master process definition. We have implemented a prototype implementation of a middleware and validated its functionality in the context of a realistic application. We have also evaluated the scalability of our solution and shown that it improves scalability characteristics when compared to the existing multi-tenant business process customization techniques. It has been also demonstrated that the performance overhead of intercepting the underlying workflow-engine is negligible. We have also shown how the middleware allows for better software management insofar as the customization issue is concerned.

This work fits into our ongoing research on customization of multi-tenant SaaS applications. In future work, we will investigate how these customizable business processes can be used to accommodate evolution, e.g. caused by changing business or tenant requirements. In addition, we will look at techniques to ensure compositional correctness of processes and sub-processes (for example, by means of explicit process typing).

8. ACKNOWLEDGMENTS

This research is partially funded by the Research Fund KU

Leuven, the ADDIS research program funded by KU Leuven GOA, D-BASE and DeCoMAdS. The D-BASE and DeCoMAdS are co-funded by iMinds (Interdisciplinary Institute for Technology), a research institute founded by the Flemish Government.

9. REFERENCES

[1] Business Process Model and Notation (BPMN). http://www.omg.org/spec/BPMN/2.0/PDF/. Accessed: 2015-08-04.

[2] RedHat JBoss jBPM. http://www.jbpm.org/. Accessed: 2015-08-04.

[3] L. Baresi, S. Guinea, and L. Pasquale. Service-oriented Dynamic Software Product Lines. *Computer*, (10):42–48, 2012.

[4] M. Braem, K. Verlaenen, N. Joncheere, W. Vanderperren, R. Van Der Straeten, E. Truyen, W. Joosen, and V. Jonckers. *Isolating Process-level Concerns Using Padus*. Springer, 2006.

[5] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. *Data & Knowledge Engineering*, 24(3):211–238, 1998.

[6] A. Charfi and M. Mezini. Aspect-oriented Web Service Composition with AO4BPEL. In *Web Services*, pages 168–182. Springer, 2004.

[7] K. Geebelen, S. Walraven, E. Truyen, S. Michiels, H. Moens, F. De Turck, B. Dhoedt, and W. Joosen. An Open Middleware for Proactive QoS-aware Service Composition in a Multi-tenant SaaS Environment. In *Proceedings of the 2012 International Conference on Internet Computing (ICOMP'12)*, 2012.

[8] F. Gey, S. Walraven, D. Van Landuyt, and W. Joosen. Building a Customizable Business-Process-as-a-Service Application with Current State-of-Practice. In *Software Composition*, pages 113–127. Springer, 2013.

[9] M. Hahn, S. Gomez Saez, V. Andrikopoulos, D. Karastoyanova, and F. Leymann. SCE^{MT}: A Multi-tenant Service Composition Engine. In *Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on*, pages 89–96. IEEE, 2014.

[10] A. Hallerbach, T. Bauer, and M. Reichert. Capturing Variability in Business Process Models: the Provop Approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(6-7):519–546, 2010.

[11] R. Hamadi and B. Benatallah. Recovery nets: Towards Self-adaptive Workflow Systems. In *Web Information Systems–WISE 2004*, pages 439–453. Springer, 2004.

[12] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented Product Line Engineering. *IEEE software*, (4):58–65, 2002.

[13] M. Koning, C.-a. Sun, M. Sinnema, and P. Avgeriou. VxBPEL: Supporting Variability for Web Services in BPEL. *Information and Software Technology*, 51(2):258–269, 2009.

[14] R. Mietzner and F. Leymann. Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 2, pages 359–366. IEEE, 2008.

[15] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability Modeling to Support Customization and Deployment of Multi-tenant-aware Software as a Service Applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25. IEEE Computer Society, 2009.

[16] R. Mietzner, T. Unger, R. Titze, and F. Leymann. Combining Different Multi-tenancy Patterns in Service-Oriented Applications. In *Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International*, pages 131–140. IEEE, 2009.

[17] S. Modafferi, E. Mussi, and B. Pernici. SH-BPEL: a Self-healing Plug-in for WS-BPEL Engines. In *Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 48–53. ACM, 2006.

[18] M. Reichert and P. Dadam. ADEPT$_{flex}$ -Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[19] S. Strauch, V. Andrikopoulos, S. G. Sáez, and F. Leymann. ESB^{MT}: A Multi-tenant Aware Enterprise Service Bus. *International Journal of Next-Generation Computing*, 4(3):230–249, 2013.

[20] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. *Science of Computer Programming*, 79:70–85, 2014.

[21] E. Truyen, N. Cardozo, S. Walraven, J. Vallejos, E. Bainomugisha, S. Günther, T. D'Hondt, and W. Joosen. Context-oriented Programming for Customizable SaaS Applications. In *Proceedings of the 27th annual acm symposium on applied computing*, pages 418–425. ACM, 2012.

[22] W. M. van der Aalst. Business Process Configuration in the Cloud: How to Support and Analyze Multi-tenant Processes? In *Web Services (ECOWS), 2011 Ninth IEEE European Conference on*, pages 3–10. IEEE, 2011.

[23] W. M. van der Aalst. Business Process Management: A Comprehensive Survey. *ISRN Software Engineering*, 2013, 2013.

[24] W. M. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270(1):125–203, 2002.

[25] S. Walraven, E. Truyen, and W. Joosen. A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications. In *Middleware 2011*, pages 370–389. Springer, 2011.

[26] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen. Efficient Customization of Multi-tenant Software-as-a-Service Applications with Service Lines. *Journal of Systems and Software*, 91:48–62, 2014.